GİRESUN ÜNİVERSİTESİ MÜHENDİSLİK FAKÜLTESİ
BİLGİSAYAR MÜHENDİSLİĞİ BÖLÜMÜ
ALGORİTMALAR DERSİ VİZE SINAVI

Adı – Soyadı – Numarası:

Soru 1: Verilen kod parçasının zaman karmaşıklığı nedir? Kısaca açıklayınız.

```
public void foo(int n) {
    for (int i = 1; i <= n; i = i * 2) {
        System.out.println("Merhaba");
    }
}
```

for döngüsü i değerini 1'den başlatarak, her adımda 2 ile çarpıyor ve n'e eşit veya küçük olana kadar devam ediyor. Her döngüde Merhaba yazdırılıyor. Döngü, i değerini 2 ile çarparak n'e ulaşana kadar $\log_2(n)$ kere tekrarlanır. Bu nedenle, kod parçasının toplam çalışma süresi n'in logaritmasına orantılıdır. Verilen kod parçasının zaman karmaşıklığı O(log n)'dir.

n = 8 ise:

1. döngüde i = 1 olur.

2. döngüde i = 2 olur.

3. döngüde i = 4 olur.

4. döngüde i = 8 olur ve döngü sona erer.

Bu örnekte, döngü 4 kere tekrarlanmıştır ve bu da $\log_2(8) + 1 = 4$'e eşittir.

Soru 2: Verilen kod parçasının zaman karmaşıklığı nedir? Kısaca açıklayınız.

```
public void foo(int[][] n) {
    for (int i = 0; i < n.length; i++) {
        for (int j = 0; j < n[0].length; j++) {
            System.out.println(n[i][j]);
        }
    }
}
```

Kod parçası, iki boyutlu bir diziyi (n) satır satır ve sütun sütun işleyerek her bir elemanı yazdırır. Bu işlem iki iç içe for döngüsü ile gerçekleştirilir: İlk döngü, i değişkeni aracılığıyla dizinin satırlarını 0'dan başlayarak n.length-1'e kadar sırayla tarar. Her bir satır için, ikinci döngü j değişkeni aracılığıyla satırdaki sütunları 0'dan başlayarak n[0].length-1'e kadar sırayla tarar. Her sütun için, n[i][j] elemanı yazdırılır.

Verilen kod parçasının zaman karmaşıklığı O(m*n)'dir. Bu, kod parçasının çalışma süresinin m*n ile orantılı olduğu anlamına gelir, burada m dizinin satır sayısını ve n ise ilk satırdaki sütun sayısını temsil eder.

Soru 3: Verilen kod parçasının zaman karmaşıklığı nedir? Kısaca açıklayınız.

```
public void foo(int m, int n) {
    for (int i = 0; i < m; i++) {
        System.out.println(i);
    }
```

Başarılar.. Dr. Öğr. Üyesi Sercan KÜLCÜ 24.04.2024

```
    for (int i = 0; i < n; i++) {
        System.out.println(i);
    }
}
```

Bu kod parçasında, iki adet for döngüsü bulunmaktadır. İlk döngü m kez çalışıyor. Döngü her bir adımda i değerini artırarak 0 ile m−1 arasındaki tamsayıları yazdırıyor. İkinci döngü n kez çalışıyor. Bu döngü de her bir adımda i değerini artırarak 0 ile n−1 arasındaki tamsayıları yazdırıyor. Her iki döngü birbirinden bağımsızdır. Dolayısıyla, bu iki döngünün çalışma zamanı birbirinden bağımsızdır.

Algoritmanın karmaşıklığı, döngülerin her birinin çalışma süresinin toplamıdır. İlk döngü m kez çalışırken, ikinci döngü n kez çalışır. Dolayısıyla, toplam karmaşıklık O(m+n) olur. Bu, m ve n değerlerinin büyüklüğü arttıkça toplam çalışma zamanının doğrusal olarak artacağı anlamına gelir.

Soru 4: Verilen kod parçasının zaman karmaşıklığı nedir? Kısaca açıklayınız.

```
public void foo(int[] n) {
    for (int i = 0; i < n.length / 2; i++) {
        System.out.println(n[i]);
    }
}
```

Bu kod parçasında, bir for döngüsü bulunmaktadır. Döngü, dizinin ilk yarısındaki elemanları yazdırmak için tasarlanmış. Bu döngü i değişkenini 0'dan başlatır ve n.length/2 'den küçük olduğu sürece çalışır. Döngünün her bir adımında, i 'inci indeksteki elemanı yazdırır. Bu durumda, döngünün çalışma zamanı dizi n'in boyutuyla doğru orantılıdır.

Zaman karmaşıklığı O(n/2) olarak ifade edilir. Ancak, büyük O gösterimi içerisinde sabit katsayılar önemsiz kabul edilir. Bu nedenle, O(n/2) ifadesi O(n) olarak basitleştirilir.

Soru 5: QuickSort algoritmasının performansında pivot seçiminin önemini kısaca açıklayınız.

```
void quickSort(int[] dizi, int baslangic, int bitis) {
    if (baslangic < bitis) {
        int pivotIndex = partition(dizi, baslangic, bitis);
        quickSort(dizi, baslangic, pivotIndex - 1);
        quickSort(dizi, pivotIndex + 1, bitis);
    }
}
```

QuickSort algoritmasının performansında pivot seçimi oldukça önemlidir çünkü pivot, algoritmanın bölme işlemini ne kadar dengeli gerçekleştireceğini belirler. Algoritma, bir pivot elemanı seçerek diziyi iki alt gruba

ayırır ve bu alt grupları tekrarlamalı olarak sıralar. İyi bir pivot seçimi, alt grupları mümkün olduğunca dengeli hale getirir. Kötü bir pivot seçimi, bir alt grubun çok küçük veya çok büyük olmasına neden olur.

Dizi [5, 2, 4, 1, 3] olsun.

Kötü Pivot: Pivot 1 olsun. Bu durumda, sol alt grupta 0 eleman ve sağ alt grupta 4 eleman olur. Bu dengesizlik algoritmanın performansını $O(n^2)$ zaman karmaşıklığına kadar düşürebilir.

İyi Pivot: Pivot 3 olsun. Bu durumda, sol alt grupta 2 ve 1 ve sağ alt grupta 4 ve 5 bulunur. Bu denge, algoritmanın daha hızlı çalışmasını sağlar.

Soru 6: İkili Arama algoritmasının karmaşıklığı nedir? Düzgün çalışması için dizi ile ilgili ön koşul nedir?

```
int binarySearch(int[] dizi, int aranan) {
    int baslangic = 0;
    int bitis = dizi.length - 1;
    while (baslangic <= bitis) {
        int orta = baslangic + (bitis - baslangic) / 2;
        if (dizi[orta] == aranan) {
            return orta;
        }
        if (dizi[orta] < aranan) {
            baslangic = orta + 1;
        } else {
            bitis = orta - 1;
        }
    }
    return -1;
}
```

Dizi n adet elemana sahip olsun. İkili arama, her bir adımda özyinelemeli olarak dizinin arama yapılacak kısmını yarıya indirir, bu nedenle dizi boyutu artsa bile arama zamanı logaritmik olarak artar. Karmaşıklığı, en kötü durumunda O(log n)'dir. Bu, algoritmanın çalışma süresinin dizinin uzunluğu (n) ile logaritmik orantıda olduğu anlamına gelir.

İkili arama algoritmasının düzgün çalışması için; dizi, küçükten büyüğe sıralanmış olmalıdır. Tekrarlanan eleman içermemelidir.

Soru 7: Kabarcık sıralama algoritması zaman karmaşıklığı nedir? Performansı nasıl geliştirilebilir?

```
void bubbleSort(int[] dizi) {
    int n = dizi.length;
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (dizi[j] > dizi[j + 1]) { // SWAP
                int gecici = dizi[j];
```

```
                dizi[j] = dizi[j + 1];
                dizi[j + 1] = gecici;
            }
        }
    }
}
```

Kabarcık sıralama algoritmasının zaman karmaşıklığı $O(n^2)$'dir. Burada, "n" dizinin elemanlarının sayısını temsil eder. İki adet iç içe döngü kullanılarak her bir elemanın diğer tüm elemanlarla karşılaştırılması sağlanır. Her bir döngü n kez döner. Dolayısıyla, her bir elemanı uygun konuma yerleştirmek n×n adımda tamamlanır.

Eğer bir adımda hiçbir değişiklik yapılmadıysa, yani dizinin sırası değişmediyse, sıralama işlemi tamamlanmış demektir. Bu durumda döngüyü erken sonlandırmak, performansı artırabilir. Diğer sıralama algoritmaları kullanılabilir.